Конспект лекции
Collections Framework

Цель и задачи лекции

Цель – изучить Collections Framework в Java.

Задачи:

1. Изучить структуру Collections Framework
2. Понять временную сложность работы структур данных
3. Описать алгоритмы работы основных классов Collections Framework

План занятия

1. Collections Framework
2. Иерархия коллекций
3. Интерфейс List
4. Интерфейс Set
5. Интерфейс Map

Collections Framework

Каркас коллекций Collections Framework в Java стандартизирует способы управления группами объектов в прикладных программах. Коллекции не были частью исходной версии языка Java, но были внедрены в версии J2SE-1.2. Каркас коллекций был разработан для достижения нескольких целей. Во-первых, он должен был обеспечивать высокую производительность. Во-вторых, каркас должен был обеспечивать единообразное функционирование коллекций с высокой степенью взаимодействия. В-третьих, коллекции должны были допускать простое расширение и/или адаптацию. В каркасе коллекций определяется несколько интерфейсов.

Java Collection Framework — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количеством структур данных из «коробки».

Основное предназначение – работа с динамическими наборами объектов. Главное отличие от массивов – динамическая размерность.

Расположение – пакет java.util, а значит требуется указывать подключаемые типы в разделе описания импортируемых типов.

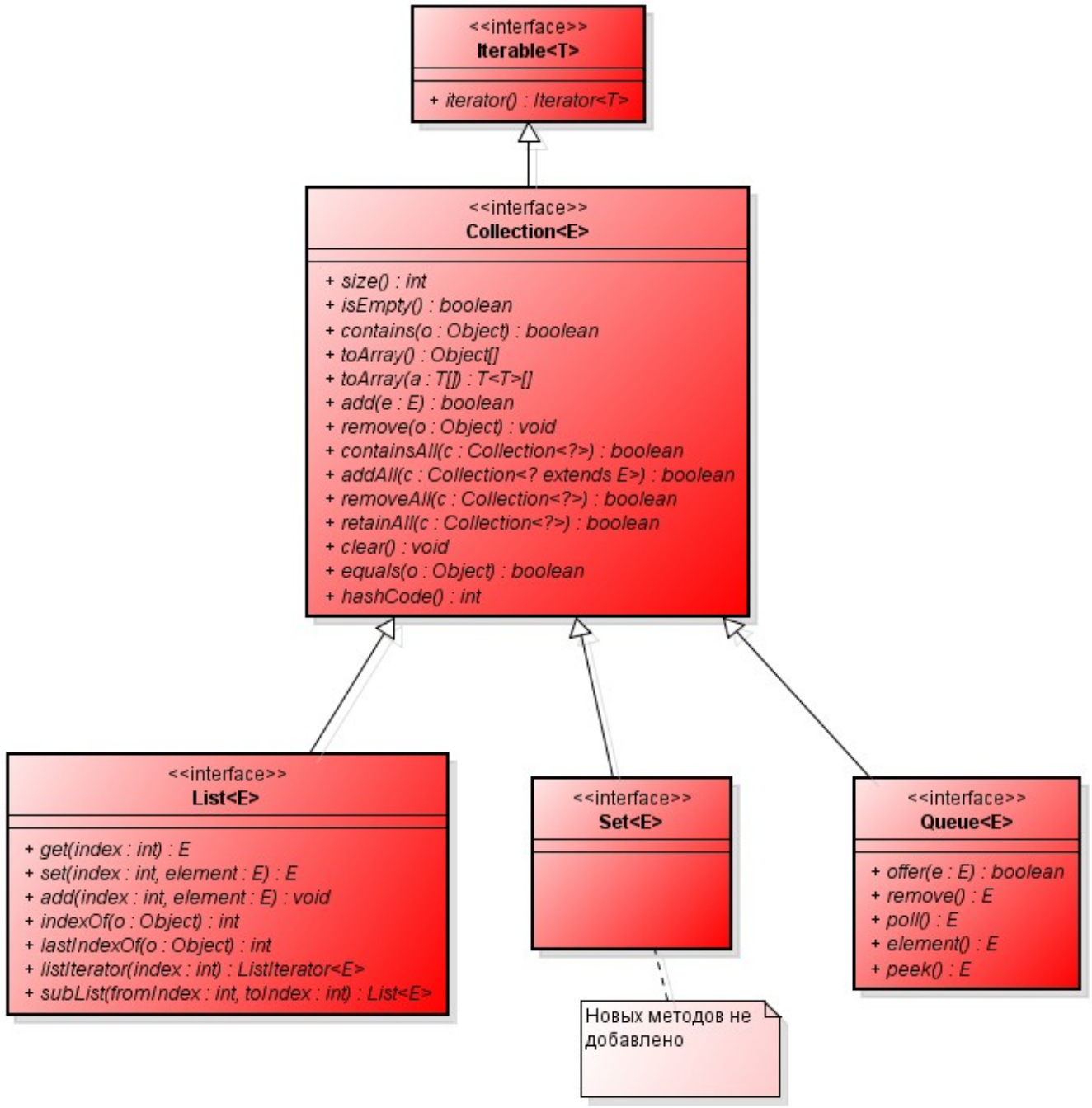
Основные концепции

В библиотеке контейнеров Java проблема хранения объектов делится на две концепции, выраженные в виде базовых интерфейсов библиотеки:

- Коллекция (Collection): группа отдельных элементов, сформированная по некоторым правилам. Класс List (список) хранит элементы в порядке вставки, в классе Set (множество) нельзя хранить повторяющиеся элементы, а класс Queue (очередь) выдает элементы в порядке, определяемом спецификой очереди (обычно это порядок вставки элементов в очередь).
- Карта (Map): набор пар объектов «ключ-значение», с возможностью выборки по ключу. ArrayList позволяет искать объекты по порядковым номерам, поэтому в каком-то смысле он связывает числа с объектами. Класс Map (карта — также встречаются термины ассоциативный массив и словарь) позволяет искать объекты по другим объектам — например, получить объект значения по объекту ключа, по аналогии с поиском определения по слову.

Иерархия коллекций

В каркасе коллекций определяется несколько и интерфейсов. Начать рассмотрение коллекций с их интерфейсов следует потому, что они определяют саму сущность классов коллекций. А конкретные классы лишь предоставляют различные реализации стандартных интерфейсов.



Интерфейсы коллекций

Интерфейс Collection расширяют интерфейсы List, Set и Queue.

1. Collection. Позволяет работать с группами объектов. Находится на вершине иерархии коллекций
2. List. Расширяет интерфейс Collection для управления последовательностями (списками объектов). Представляет собой не упорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
3. NavigableSet. Расширяет интерфейс SortedSet для извлечения элементов по результатам поиска ближайшего совпадения

4. Queue. Расширяет интерфейс Collection для управления специальными типами списков, где элементы удаляются только из начала список. Это коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.
5. Deque. Расширяет интерфейс Queue для организации двусторонних очередей
6. Set. Расширяет интерфейс Collection для управления множествами, которые должны содержать однозначные элементы. описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).
7. SortedSet. Расширяет интерфейс Set для управления отсортированными множествами.

Помимо перечисленных выше и интерфейсов, для составления коллекций используются интерфейсы Comparator, RandomAccess, Iterator и ListIterator, которые подробнее рассматриваются далее в этой главе. А начиная с версии JDK 8 к их числу принадлежит также и интерфейс Spliterator. Если говорить кратко, то интерфейс Comparator определяет два сравниваемых объекта, а интерфейсы Iterator, ListIterator и Spliterator перечисляют объекты в коллекции. Если же список реализует интерфейс RandomAccess, то тем самым он поддерживает эффективный произвольный доступ к своим элементам.

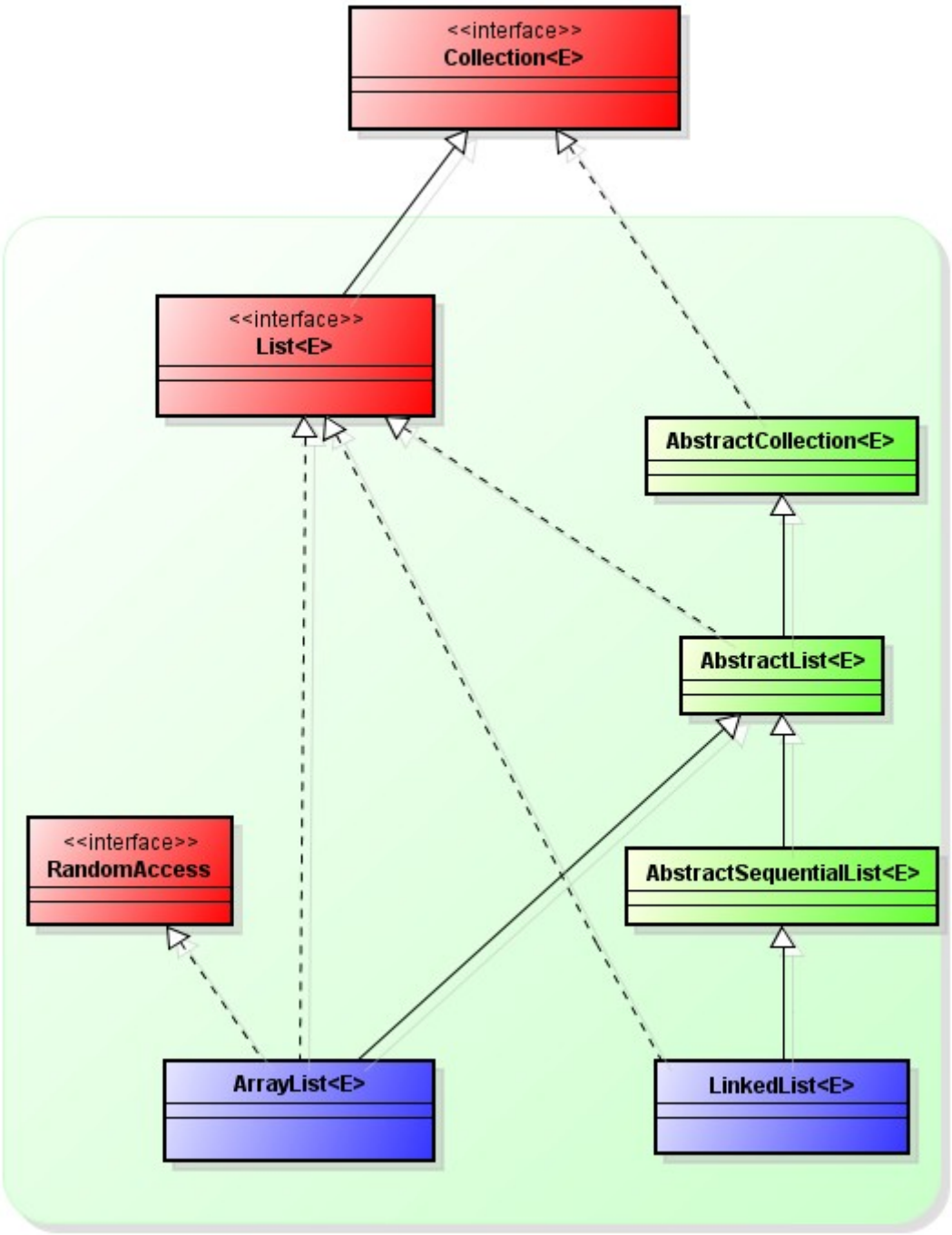
Интерфейс List

Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.

List добавляет следующие методы:

1. void add(int index, E obj) - вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются.
2. boolean addAll (int index,Collection<? extends E> c) - вставляет все элементы в вызывающий список, начиная с позиции, переданной в index. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает true, если вызывающий список изменяется, и false в противном случае.
3. E get (int index) - возвращает объект, сохраненный в указанной позиции вызывающего списка.
4. int indexOf(Object obj) - возвращает индекс первого экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.

5. int lastIndexOf(Object obj) - возвращает индекс последнего экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.
6. ListIterator listIterator() - возвращает итератор, указывающий на начало списка.
7. ListIterator listIterator(int index) - возвращает итератор, указывающий на заданную позицию в списке.
8. E remove(int index) - удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.
9. E set (int index, E obj) - присваивает obj элементу, находящемуся в списке в позиции index.
10. default void sort(Comparator<? super E> c) - сортирует список, используя заданный компаратор (добавлен в версии JDK 8).
11. List subList (int start, int end) - возвращает список, включающий элементы от start до end-1 из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.



Класс ArrayList

Класс ArrayList реализует интерфейс List. ArrayList поддерживает динамические массивы, которые могут расти по мере необходимости. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

Объект класса ArrayList, содержит свойства elementData и size. Хранилище значений elementData есть ни что иное как массив определенного типа (указанного в generic).

У этого класса есть следующие конструкторы:

```
ArrayList()
```

```
ArrayList(Collection <? extends E> c)

ArrayList(int capacity)
```

Достоинства класса ArrayList:

- Быстрый доступ по индексу.
- Быстрая вставка и удаление элементов с конца.

Недостатком класса ArrayList считается медленная вставка и удаление элементов в середину, однако в последних версиях эти операция были оптимизированы.

Методы класса ArrayList для добавления элементов:

- boolean add(E obj) - добавляет obj к вызывающей коллекции. Возвращает true, если obj был добавлен к коллекции. (Интерфейс Collection)
- void add(int index, E obj) - вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются. (Интерфейс List)
- E set (int index, E obj) - присваивает obj элементу, находящемуся в списке в позиции index. (Интерфейс List)
- boolean addAll (Collection<? extends E> c) - добавляет все элементы к вызывающей коллекции. Возвращает true, если операция удалась (то есть все элементы добавлены). В противном случае возвращает false. (Интерфейс Collection)
- Методы класса ArrayList для удаления элементов:
- boolean remove(Object obj) - удаляет один экземпляр obj из вызывающей коллекции. Возвращает true, если элемент удален. В противном случае возвращает false. (Интерфейс Collection)
- E remove(int index) - удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад. (Интерфейс List)
- boolean removeAll(Collection<?> c) - удаляет все элементы из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает false. (Интерфейс Collection)
- boolean retainAll(Collection<?> c) - удаляет все элементы кроме входящих из вызывающей коллекции. Возвращает true, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает false. (Интерфейс Collection)
- void clear() - удаляет все элементы вызывающей коллекции. (Интерфейс Collection)

Обратим внимание на операцию добавления элемента. Внутри метода add(value) происходят следующие вещи:

1. проверяется, достаточно ли места в массиве для вставки нового элемента;

```
ensureCapacity(size + 1);
```

2. добавляется элемент в конец (согласно значению size) массива.

elementData[size++] = element;

Остановимся только на основных моментах метода ensureCapacity(minCapacity). Если места в массиве не достаточно, новая емкость рассчитывается по формуле (oldCapacity * 3) / 2 + 1. Второй момент это копирование элементов. Оно осуществляется с помощью native метода System.arraycopy(), который написан не на Java.

```
// newCapacity - новое значение емкости
elementData = (E[])new Object[newCapacity];

// oldData - временное хранилище текущего массива с данными
System.arraycopy(oldData, 0, elementData, 0, size);
```

Добавление элемента на позицию с определенным индексом происходит в три этапа:

1. проверяется, достаточно ли места в массиве для вставки нового элемента;

```
ensureCapacity(size+1);
```

2. подготавливается место для нового элемента с помощью System.arraycopy();

```
System.arraycopy(elementData, index, elementData, index + 1, size - index);
```

3. перезаписывается значение у элемента с указанным индексом.

```
elementData[index] = element;

size++;
```

В случаях, когда происходит вставка элемента по индексу и при этом в вашем массиве нет свободных мест, то вызов System.arraycopy() случится дважды: первый в ensureCapacity(), второй в самом методе add(index, value), что явно скажется на скорости всей операции добавления.

В случаях, когда в исходный список необходимо добавить другую коллекцию, да еще и в «середину», стоит использовать метод addAll(index, Collection). И хотя, данный метод скорее всего вызовет System.arraycopy() три раза, в итоге это будет гораздо быстрее поэлементного добавления.

Удалять элементы можно двумя способами:

- по индексу remove(index)
- по значению remove(value)

С удалением элемента по индексу всё достаточно просто при вызове метода remove(index):

list.remove(5);

1. Сначала определяется какое количество элементов надо скопировать

```
int numMoved = size - index - 1;
```

2. затем копируем элементы используя System.arraycopy()

```
System.arraycopy(elementData, index + 1, elementData, index, numMoved);
```

3. уменьшаем размер массива и забываем про последний элемент

```
elementData[--size] = null; // Let gc do its work
```

При удалении по значению, в цикле просматриваются все элементы списка, до тех пор, пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.

Класс LinkedList

Этот класс расширяет класс AbstractSequentialList<E> и реализует интерфейсы

List<E>, Deque<E>. Он предоставляет структуру данных связного списка.

Класс LinkedList является обобщенным и объявляется следующим образом:

```
class LinkedList<E>
```

где E обозначает тип сохраняемых в списке объектов. У класса LinkedList имеются два конструктора:

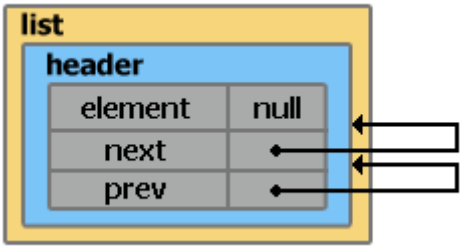
```
LinkedList()

LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список, а второй - связный список и инициирует его содержимым коллекции c. В классе LinkedList реализуется интерфейс Deque, и благодаря этому становятся доступными методы, определенные в интерфейсе Deque.

Только что созданный объект list, содержит свойства header и size.

header — псевдо-элемент списка. Его значение всегда равно null, а свойства next и prev всегда указывают на первый и последний элемент списка соответственно. Так как на данный момент список еще пуст, свойства next и prev указывают сами на себя (т.е. на элемент header). Размер списка size равен 0.

header.next = header.prev = header;



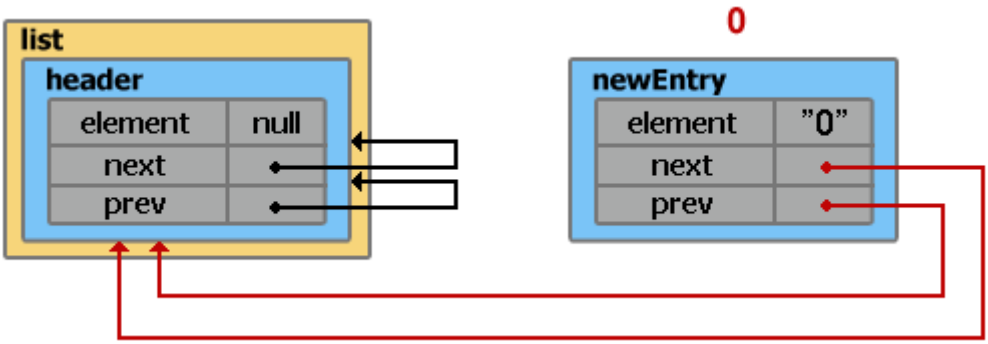
Добавление элемента в конец списка с помощью методом add(value), addLast(value) и добавление в начало списка с помощью addFirst(value) выполняется за время O(1).

Внутри класса LinkedList существует static inner класс Entry, с помощью которого создаются новые элементы.

Каждый раз при добавлении нового элемента, по сути выполняется два шага:

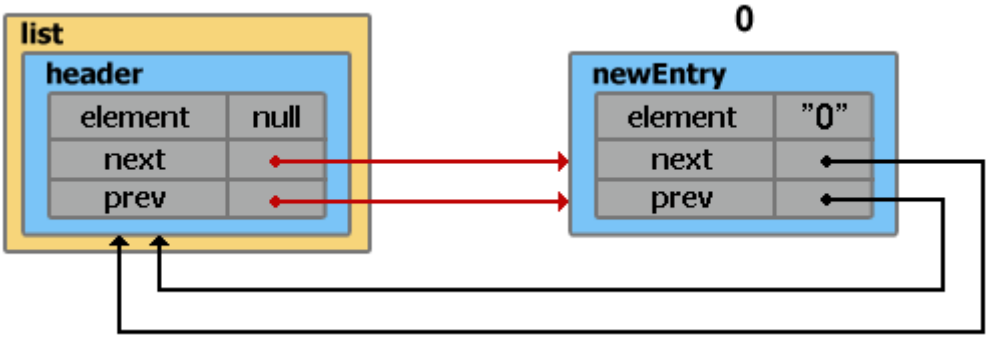
1. создается новый новый экземпляр класса Entry

```
Entry newEntry = new Entry("0", header, header.prev);
```



2. переопределяются указатели на предыдущий и следующий элемент

```
newEntry.prev.next = newEntry;

newEntry.next.prev = newEntry;

size++;
```



Для того чтобы добавить элемент на определенную позицию в списке, необходимо вызвать метод add(index, value). Отличие от add(value) состоит в определении элемента перед которым будет производиться вставка

```
(index == size ? header : entry(index))
```

Метод entry(index) пробегает по всему списку в поисках элемента с указанным индексом. Направление обхода определяется условием (index < (size >> 1)). По факту получается что для нахождения нужного элемента перебирается не больше половины списка, но с точки зрения асимптотического анализа время на поиск растет линейно — O(n).

Удалять элементы из списка можно несколькими способами:

- из начала или конца списка с помощью removeFirst(), removeLast() за время O(1);
- по индексу remove(index) и по значению remove(value) за время O(n).

Интерфейс ListIterator

ListIterator расширяет интерфейс Iterator для двустороннего обхода списка и видоизменения его элементов. ListIterator можно получить вызывая метод listIterator() для коллекций, реализующих List.

```
ListIterator<String> itr = list.listIterator();
```

Приведенный выше код помещает указатель в начало списка. Также можно начать перебор элементов с определенного места, для этого нужно передать индекс в метод listIterator(index). В случае, если необходимо начать обход с конца списка, можно воспользоваться методом descendingIterator().

Стоит помнить, что ListIterator может вызвать исключение ConcurrentModificationException, если после создания итератора, список был изменен не через собственные методы итератора.

Пример перебора элементов с использованием итератора:

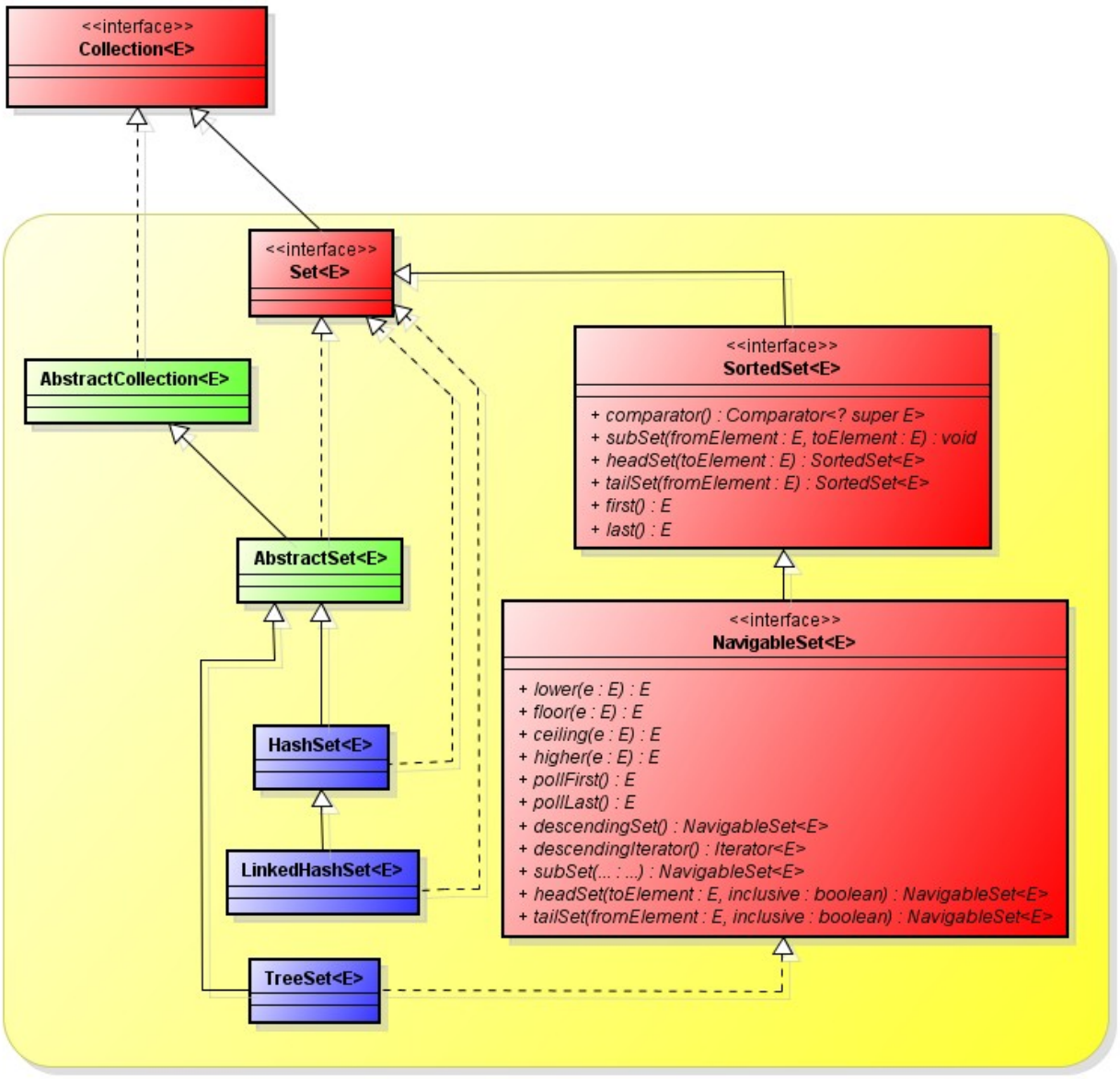
```
while (itr.hasNext())

    System.out.println(itr.next());
```

Интерфейс Set

В интерфейсе Set определяется множество. Он расширяет интерфейс Collection и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод add () возвращает логическое значение false при попытке ввести в множество дублирующий элемент. В этом интерфейсе не определяется никаких дополнительных методов.

Так как Set является интерфейсом, вам необходимо создать экземпляр с конкретной реализацией интерфейса для того, чтобы его использовать. В API Java Collections есть следующие реализации интерфейса Set:

- java.util.EnumSet;
- java.util.HashSet;
- java.util.LinkedHashSet;
- java.util.TreeSet.



Интерфейс SortedSet

Интерфейс SortedSet расширяет интерфейс Set и определяет поведение множеств, отсортированных в порядке возрастания. Интерфейс SortedSet является обобщенным и объявляется приведенным ниже образом, где E обозначает тип объектов, которые должно содержать множество:

```
interface SortedSet<E>
```

В интерфейсе SortedSet определен ряд методов, упрощающих обработку элементов множеств. Чтобы получить первый элемент в отсортированном множестве, достаточно вызвать метод first() , а чтобы получить последний элемент - метод last() .

Интерфейс NavigaЬleSet

Этот интерфейс расширяет интерфейс SortedSet и определяет поведение коллекции, извлечение элементов из которой осуществляется на основании наиболее точного совпадения с заданным значением или несколькими значениями. Интерфейс NavigableSet является обобщенным и объявляется следующим образом:

```
interface NavigaЬleSet<E>
```

где E обозначает тип объектов, содержащихся в множестве. Помимо методов, наследуемых из интерфейса SortedSet.

Класс HashSet

Класс HashSet расширяет класс AbstractSet и реализует интерфейс Set. Он служит для создания коллекции, для хранения элементов которой используется хеш-таблица. Класс HashSet является обобщенным и объявляется приведенным ниже образом, где E обозначает тип объектов, которые будут храниться в хеш-множестве. HashSet инкапсулирует в себе объект HashMap (то-есть использует для хранения хэш-таблицу).

```
class HashSet<E>
```

Для хранения данных в хеш-таблице применяется механизм так называемого хеширования, где содержимое ключа служит для определения однозначного значения, называемого хеш-кодам. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом. Преобразование ключа в хеш-код выполняется автоматически, хотя сам хеш-код недоступен. Кроме того, в прикладном коде нельзя индексировать хеш-таблицу непосредственно. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов add( ), contains( ), remove( ) и size ( ) - даже для крупных множеств.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

- HashSet(): создает пустой список
- HashSet(Collection<? extends E> col): создает хеш-таблицу, в которую добавляет все элементы коллекции col
- HashSet(int capacity): параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16

- HashSet(int capacity, float koef): параметр koef или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение.

Пример использования HashSet:

```
public class CreateHashSetExample {
    public static void main(String[] args) {
        // Creating a HashSet
        Set<String> daysOfWeek = new HashSet<>();

        // Adding new elements to the HashSet
        daysOfWeek.add("Monday");
        daysOfWeek.add("Tuesday");
        daysOfWeek.add("Wednesday");
        daysOfWeek.add("Thursday");
        daysOfWeek.add("Friday");
        daysOfWeek.add("Saturday");
        daysOfWeek.add("Sunday");

        // Adding duplicate elements will be ignored
        daysOfWeek.add("Monday");

        System.out.println(daysOfWeek);
    }
}
```

На экран будет выведено:

[Monday, Thursday, Friday, Sunday, Wednesday, Tuesday, Saturday]

Для хранения объектов пользовательских классов в HashSet, нужно переопределить методы hashCode() и equals(), иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который скорее-всего вернет разный хэш-код для ваших объектов).

Важно отметить, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов.

Класс LinkedHashSet

Класс LinkedHashSet расширяет класс HashSet, не добавляя никаких новых методов. Этот класс является обобщенным и объявляется следующим образом:

```
class LinkedBashSet<E>
```

где E обозначает тип объектов, которые будут храниться в хеш-множестве. У этого класса такие же конструкторы, как и у класса HashSet.

В классе LinkedHashSet поддерживается связный список элементов хеш-множества в том порядке, в каком они введены в него. Это позволяет

организовать итерацию с вводом элементов в определенном порядке. Следовательно, когда перебор элементов хеш-множества типа LinkedHashSet производится с помощью итератора, элементы извлекаются из этого множества в том порядке, в каком они были введены. Именно в этом порядке они будут также возвращены методом toString(), вызываемым для объекта типа LinkedHashSet.

Класс TreeSet

Класс TreeSet – реализует интерфейс NavigableSet<E>, который поддерживает элементы в отсортированном по возрастанию порядке. Объекты сохраняются в отсортированном порядке по нарастающей. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

В классе HashSet определены следующие конструкторы:

```
TreeSet()

TreeSet(Collection<? extends E> collection)

TreeSet(Comparator<? super E> comparator)

TreeSet(SortedSet<E> sortedSet)
```

TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n).

Интерфейс Queue

Интерфейс Queue расширяет Collection и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел первый вышел“ (FIFO). Существуют разные типы очередей, в которых порядок основан на некотором критерии. Очереди не могут хранить значения null.

Методы интерфейса Queue:

- E element() - возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, инициируется исключение NoSuchElementException.
- E remove() - удаляет элемент из головы очереди, возвращая его. Инициирует исключение NoSuchElementException, если очередь пуста.
- E peek() - возвращает элемент из головы очереди. Возвращает null, если очередь пуста. Элемент не удаляется.
- E poll() - возвращает элемент из головы очереди и удаляет его. Возвращает null, если очередь пуста.
- boolean offer(E obj) - пытается добавить obj в очередь. Возвращает true, если obj добавлен, и false в противном случае.

Интерфейс Deque

Интерфейс Deque появился в Java 6. Он расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь FIFO либо как стек LIFO.

Методы интерфейса Deque:

- void addFirst(E obj) - добавляет obj в голову двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.
- void addLast(E obj) - добавляет obj в хвост двунаправленной очереди. Возбуждает исключение IllegalStateException, если в очереди ограниченной емкости нет места.
- E getFirst() - возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение NoSuchElementException.
- E getLast() - возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключения NoSuchElementException.
- boolean offerFirst(E obj) - пытается добавить obj в голову двунаправленной очереди. Возвращает true, если obj добавлен, и false в противном случае. Таким образом, этот метод возвращает false при попытке добавить obj в полную двунаправленную очередь ограниченной емкости.
- boolean offerLast(E obj) - пытается добавить obj в хвост двунаправленной очереди. Возвращает true, если obj добавлен, и false в против ном случае.
- E pop() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.
- void push(E obj) - добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированного объема нет места, возбуждает исключение IllegalStateException.
- E peekFirst() - возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.
- E peekLast() - возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает null, если очередь пуста. Объект из очереди не удаляется.
- E pollFirst() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.
- E pollLast() - возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает null, если очередь пуста.
- E removeLast() - возвращает элемент, находящийся в конце двунаправленной очереди, удаляя его в процессе. Возбуждает исключение NoSuchElementException, если очередь пуста.

- E removeFirst() - возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение NoSuchElementException, если очередь пуста.
- boolean removeLastOccurrence(Object obj) - удаляет последнее вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false если очередь не содержала obj.
- boolean removeFirstOccurrence(Object obj) - удаляет первое вхождение obj из двунаправленной очереди. Возвращает true в случае успеха и false, если очередь не содержала obj.

Класс ArrayDeque

ArrayDeque создает двунаправленную очередь.

Конструкторы класса ArrayDeque:

- ArrayDeque() - создает пустую двунаправленную очередь с вместимостью 16 элементов.
- ArrayDeque(Collection<? extends E> c) - создает двунаправленную очередь из элементов коллекции c в том порядке, в котором они возвращаются итератором коллекции c.
- ArrayDeque(int numElements) - создает пустую двунаправленную очередь с вместимостью numElements.

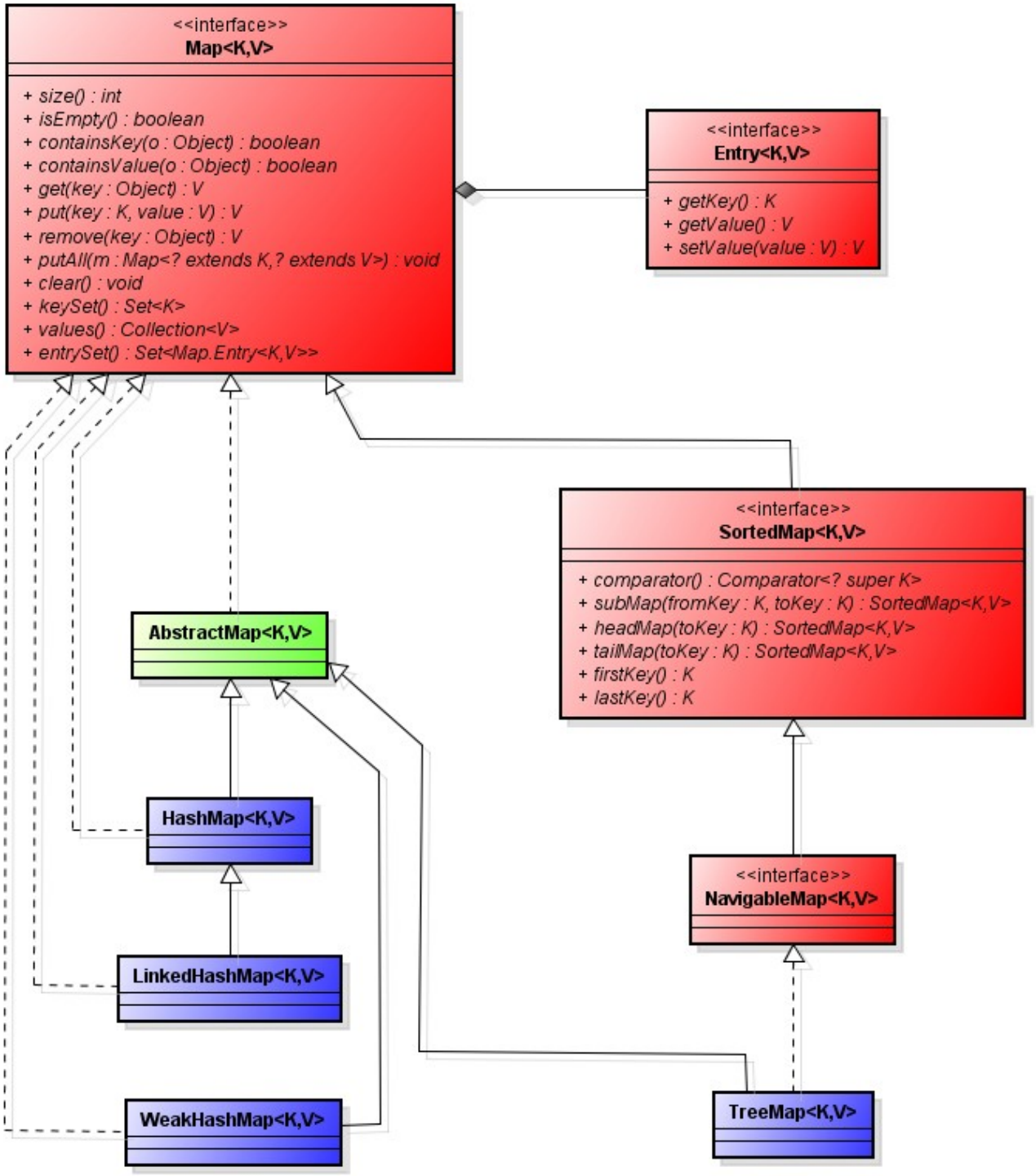
Интерфейсы отображений

Интерфейсы отображений определяют их характер и особенности. Рассмотрим интерфейсы отображений:

- Map – отображает однозначные ключи на значения
- Map.Entry – описывает элемент отображения (пару «ключ-значение»). Это внутренний класс интерфейса Map
- SortedMap – расширяет интерфейс Map таким образом, чтобы ключи располагались по нарастающей
- NavigableMap – расширяет интерфейс SortedMap таким образом, чтобы ключи располагались по нарастающей

Интерфейс Map

Интерфейс Map<K, V> представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.



Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map не расширяет интерфейс Collection, но считается частью Collection Framework.

Среди методов интерфейса Map можно выделить следующие:

- void clear(): очищает коллекцию
- boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k
- boolean containsValue(Object v): возвращает true, если коллекция содержит значение v

- Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry
- boolean equals(Object obj): возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj
- boolean isEmpty: возвращает true, если коллекция пуста
- V get(Object k): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null
- V getOrDefault(Object k, V defaultValue): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение defaultValue
- V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null
- V putIfAbsent(K k, V v): помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом
- Set<K> keySet(): возвращает набор всех ключей отображения
- Collection<V> values(): возвращает набор всех значений отображения
- void putAll(Map<? extends K, ? extends V> map): добавляет в коллекцию все объекты из отображения map
- V remove(Object k): удаляет объект с ключом k
- int size(): возвращает количество элементов коллекции

Обращение с отображениями опирается на две основные операции, выполняемые методами get() и put() . Чтобы ввести значение в отображение, следует вызвать метод put() , указав ключ И значение, а для того чтобы получить значение из отображения - вызвать метод get(), передав ему ключ в качестве аргумента. По этому ключу будет возвращено связанное с ним значение.

Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов Map.Entry<K, V>.

HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени.

LinkedHashMap - расширяет класс HashMap. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса LinkedHashMap, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса LinkedHashMap, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.

TreeMap - расширяет класс AbstractMap и реализует интерфейс NavigatebleMap. Он создает коллекцию, которая для хранения элементов

применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.

WeakHashMap - коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка (англ. weak reference) — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными».

Интерфейс Map.Entry

Обобщенный интерфейс Map.Entry<K, V> представляет объект с ключом типа K и значением типа V и определяет следующие методы:

- boolean equals(Object obj): возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему
- K getKey(): возвращает ключ объекта отображения
- V getValue(): возвращает значение объекта отображения
- Set<K> keySet(): возвращает набор всех ключей отображения
- V setValue(V v): устанавливает для текущего объекта значение v
- int hashCode(): возвращает хеш-код данного объекта

Класс HashMap

Класс HashMap реализует интерфейс Map. Он использует хеш-таблицу для хранения карты. Это позволяет обеспечить константное время выполнения методов get() и put() даже при больших наборах. Ключи и значения могут быть любых типов, в том числе и null.

Конструкторы класса HashMap:

- HashMap() - Конструктор с используемыми по умолчанию значениями начальной емкости (16) и коэффициентом загрузки (0.75)
- HashMap(int initialCapacity) - Конструктор с используемым по умолчанию значением коэффициентом загрузки (0.75) и начальной емкостью initialCapacity
- HashMap(int initialCapacity, float loadFactor) - Конструктор с используемыми значениями начальной емкости initialCapacity и коэффициентом загрузки loadFactor
- HashMap(Map<? extends K,? extends V> m) - Конструктор с определением структуры согласно объекту-параметра.

Хеширование

Хэширование -это процесс преобразования объекта в целочисленную форму, выполняется с помощью метода hashCode(). Очень важно правильно реализовать метод hashCode() для обеспечения лучшей производительности класса HashMap.

Здесь используется свой собственный класс Key и таким образом переопределяется метод hashCode() для демонстрации различных сценариев. Рассмотрим класс Key:

```
// специальный класс Key для переопределения методов hashCode()
// и equals()
class Key {
    String key;
    Key(String key) {
        this.key = key;
    }

    @Override
    public int hashCode() {
        return (int)key.charAt(0);
    }

    @Override
    public boolean equals(Object obj) {
        return key.equals((String)obj);
    }
}
```

Здесь переопределенный метод hashCode() возвращает ASCII код первого символа строки. Таким образом, если первые символы строки одинаковые, то и хэш коды будут одинаковыми. Не стоит использовать подобную логику в своих программах.

Этот код создан исключительно для демонстрации. Поскольку HashCode допускает ключ типа null, хэш код null всегда будет равен 0.

Метод hashCode() используется для получения хэш кода объекта. Метод hashCode() класса Object возвращает ссылку памяти объекта в целочисленной форме (идентификационный хеш (identity hash code)). Сигнатура метода public native hashCode(). Это говорит о том, что метод реализован как нативный, поскольку в java нет какого -то метода позволяющего получить ссылку на объект. Допускается определять собственную реализацию метода hashCode(). В классе HashMap метод hashCode() используется для вычисления корзины (bucket) и следовательно вычисления индекса.

Метод equals используется для проверки двух объектов на равенство. Метод реализованн в классе Object. Вы можете переопределить его в своем собственном классе. В классе HashMap метод equals() используется для проверки равенства ключей. В случае, если ключи равны, метод equals() возвращает true, иначе false.

Внутреннее устройство HashMap

Bucket - это единственный элемент массива HashMap. Он используется для хранения узлов (Nodes). Два или более узла могут иметь один и тот -же bucket. В этом случае для связи узлов используется структура данных связанный список. Bucket -ы различаются по ёмкости (свойство capacity). Отношение между bucket и capacity выглядит следующим образом:

```
capacity = number of buckets * load factor
```

Один bucket может иметь более, чем один узел, это зависит от реализации метода hashCode(). Чем лучше реализован ваш метод hashCode(), тем лучше будут использоваться ваши bucket-ы.

Рассмотрим, как происходит вычисление индекса в HashMap.

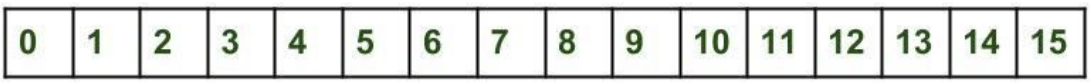
Хэш код ключа может быть достаточно большим для создания массива. Сгенерированный хэш код может быть в диапазоне целочисленного типа и если мы создадим массив такого размера, то легко получим исключение outOfMemoryException. Потому мы генерируем индекс для минимизации размера массива. По сути, для вычисления индекса выполняется следующая операция:

```
index = hashCode(key) & (n-1).
```

где n равна числу bucket или значению длины массива. В нашем примере рассматривается n, как значение по умолчанию равное 16.

Изначально HashMap пустой и размер его равен 16:

```
HashMap map = new HashMap();
```



Алгоритм добавления элементов HashMap

Рассмотрим следующий HashMap и добавление элемента в него:

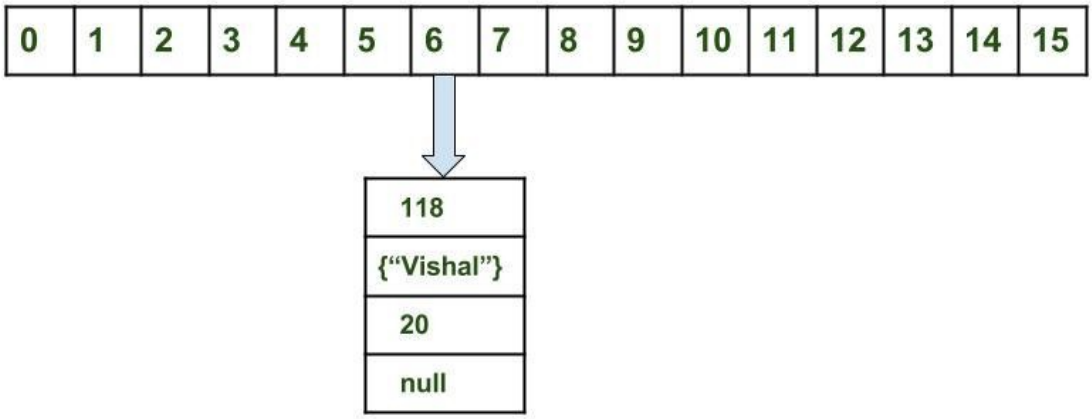
```
HashMap<Key, Integer> map = new HashMap<>();
map.put(new Key("vishal"), 20);
```

Для добавления элемента в HashMap происходит несколько шагов:

1. Вычислить значение ключа {"vishal"}. Оно будет сгенерировано, как 118.
2. Вычислить индекс с помощью метода index, который будет равен 6.
3. Создать объект node.

```
{
    int hash = 118

    // {"vishal"} не строка, а
    // объект класса Key
    Key key = {"vishal"}

    Integer value = 20
    Node next = null
}
```

4. Поместить объект в позицию с индексом 6, если место свободно.



Рассмотрим добавление еще одного элемента в HashMap:

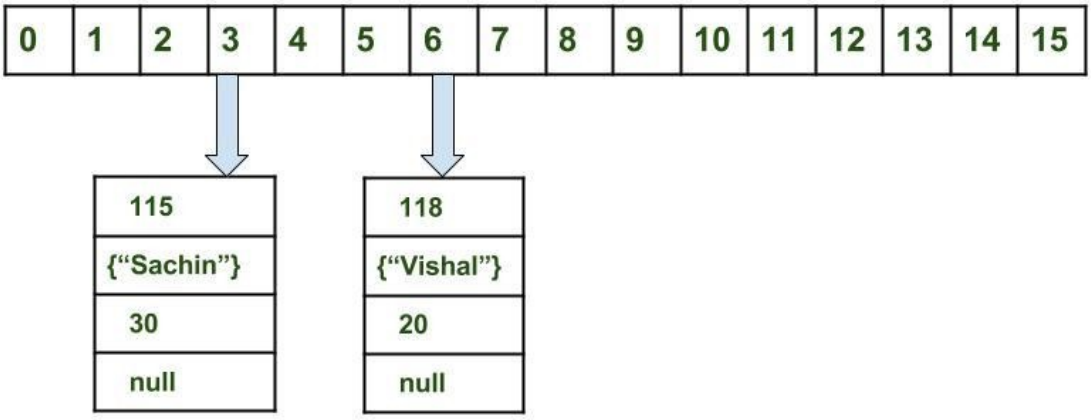
```
map.put(new Key("sachin"), 30);
```

Шаги:

1. Вычислить значение ключа {"sachin"}. Оно будет сгенерировано, как 115.
2. Вычислить индекс с помощью метода index, который будет равен 3.
3. Создать объект node.

```
{
    int hash = 115
    Key key = {"sachin"}
    Integer value = 30
    Node next = null
}
```

4. Поместить объект в позицию с индексом 3, если место свободно.



Рассмотрим случай возникновения коллизий, добавим другую пару:

```
map.put(new Key("vaibhav"), 40);
```

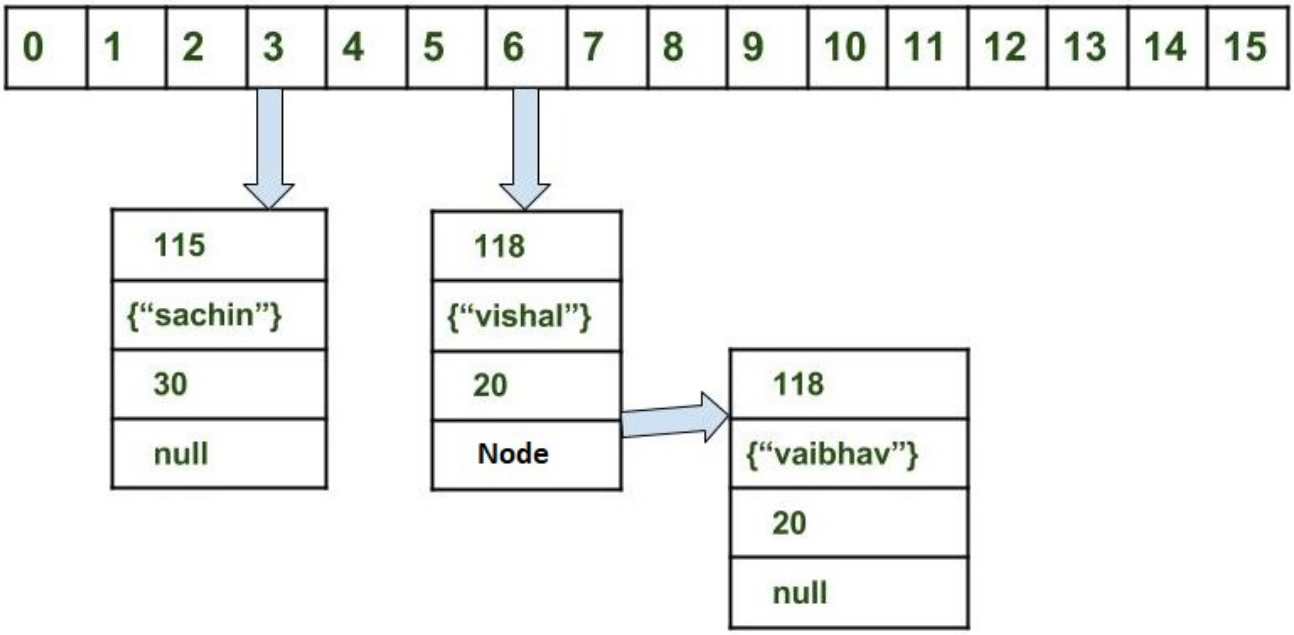
Шаги:

1. Вычислить значение ключа {"vaibhav"}. Оно будет сгенерировано, как 118.
2. Вычислить индекс с помощью метода index, который будет равен 6.
3. Создать объект node.

```
{
    int hash = 118
    Key key = {"vaibhav"}
    Integer value = 20
    Node next = null
}
```

4. Поместить объект в позицию с индексом 6, если место свободно.
5. В данном случае в позиции с индексом 6 уже существует другой объект, этот случай называется коллизией.
6. В таком случае проверяется с помощью методов hashCode() и equals(), что оба ключа одинаковы.
7. Если ключи одинаковы, заменить текущее значение новым.
8. Иначе связать новый и старый объекты с помощью структуры данных "связанный список", указав ссылку на следующий объект в текущем и сохранить оба под индексом 6.

Теперь HashMap выглядит примерно так:



Алгоритм работы метода получения элементов HashMap

Рассмотрим получение значений по ключу "sachin":

```
map.get(new Key("sachin"));
```

Шаги:

1. Вычислить хэш код объекта {“sachin”}. Он был сгенерирован, как 115.
2. Вычислить индекс с помощью метода index, который будет равен 3.

3. Перейти по индексу 3 и сравнить ключ первого элемента с имеющемся значением. Если они равны – вернуть значение, иначе выполнить проверку для следующего элемента, если он существует.
4. В нашем случае элемент найден и возвращаемое значение равно 30.

Рассмотрим получение значений по ключу "vaibahv":

```
map.get(new Key("vaibhav"));
```

Шаги:

1. Вычислить хэш код объекта {"vaibhav"}. Он был сгенерирован, как 118.
2. Вычислить индекс с помощью метода index, который будет равен 6.
3. Перейти по индексу 6 и сравнить ключ первого элемента с имеющемся значением. Если они равны – вернуть значение, иначе выполнить проверку для следующего элемента, если он существует.
4. В данном случае он не найден и следующий объект node не равен null.
5. Если следующий объект node равен null, возвращаем null.
6. Если следующий объект node не равен null, переходим к нему и повторяем первые три шага до тех пор, пока элемент не будет найден или следующий объект node не будет равен null.

Пример работы HashMap:

```
import java.util.HashMap;

class Key {
    String key;
    Key(String key) {
        this.key = key;
    }

    @Override
    public int hashCode() {
        int hash = (int)key.charAt(0);
        System.out.println("hashCode for key: "
                + key + " = " + hash);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        return key.equals(((Key)obj).key);
    }
}

public class Main {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put(new Key("vishal"), 20);
        map.put(new Key("sachin"), 30);
        map.put(new Key("vaibhav"), 40);

        System.out.println();
        System.out.println("Value for key sachin: " + map.get(new Key("sachin")));
        System.out.println("Value for key vaibhav: " + map.get(new Key("vaibhav")));
    }
}
```

Вывод:

```
hashCode for key: vishal = 118
hashCode for key: sachin = 115
hashCode for key: vaibhav = 118

hashCode for key: sachin = 115
Value for key sachin: 30
hashCode for key: vaibhav = 118
Value for key vaibhav: 40
```

В Java 8 произошли изменения во внутреннем устройстве HashMap. Как мы уже знаем в случае возникновения коллизий объект node сохраняется в структуре данных "связанный список" и метод equals() используется для сравнения ключей. Это сравнения для поиска верного ключа в связанном списке -линейная операция и в худшем случае сложность равна O(n).

Для исправления этой проблемы в Java 8 после достижения определенного порога вместо связанных списков используются сбалансированные деревья. Это означает, что HashMap в начале сохраняет объекты в связанном списке, но после того, как количество элементов в хэше достигает определенного порога происходит переход к сбалансированным деревьям. Что улучшает производительность в худшем случае с O(n) до O(log n).

Следует отметить, что:

1. Сложность операций get() и put() практически константна до тех пор, пока не будет проведено повторное хэширование.
2. В случае коллизий, если индексы двух и более объектов node одинаковые, объекты node соединяются с помощью связанного списка, т.е. ссылка на второй объект node хранится в первом, на третий во втором и т.д.
3. Если данный ключ уже существует в HashMap, значение перезаписывается.
4. Хэш код null равен 0.
5. Когда объект получается по ключу происходят переходы по связанному списку до тех пор, пока объект не будет найден или ссылка на следующий объект не будет равна null.

Класс LinkedHashMap

Конструкторы класса LinkedHashMap:

- LinkedHashMap() - Конструктор с используемыми по умолчанию значениями начальной емкости (16) и коэффициентом загрузки (0.75)
- LinkedHashMap(int initialCapacity) - Конструктор с используемым по умолчанию значением коэффициентом загрузки (0.75) и начальной емкостью initialCapacity
- LinkedHashMap(int initialCapacity, float loadFactor) - Конструктор с используемыми значениями начальной емкости initialCapacity и коэффициентом загрузки loadFactor

- LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) - Конструктор с используемыми значениями начальной емкости initialCapacity и коэффициентом загрузки loadFactor. accessOrder - the ordering mode - true for access-order, false for insertion-order
- LinkedHashMap(Map<? extends K,? extends V> m) - Конструктор с определением структуры согласно объекта-параметра.

Класс TreeMap

Класс TreeMap расширяет класс AbstractMap и реализует интерфейс NavigatebleMap. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.

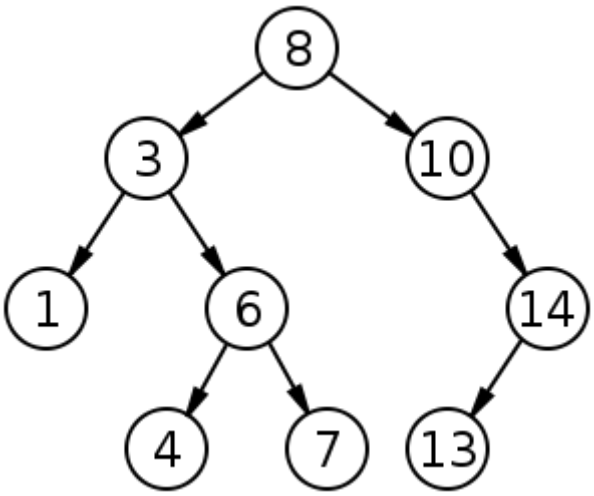
Конструкторы класса TreeMap:

- TreeMap() - Пустой конструктор создания объекта без упорядочивания данных (using the natural ordering of its keys)
- TreeMap(Comparator<? super K> comparator) - Конструктор создания объекта с упорядочиванием значений согласно comparator
- TreeMap(Map<? extends K,? extends V> m) - Конструктор с определением структуры согласно объекту-параметра
- TreeMap(SortedMap<K,? extends V> m) - Конструктор с определением структуры и сортировки согласно объекту-параметра.

TreeMap основан на Красно-Черном дереве, вследствие чего TreeMap сортирует элементы по ключу в естественном порядке или на основе заданного вами компаратора.TreeMap гарантирует скорость доступа log(n) для операций containsKey, get, put и remove.

Двоичное дерево поиска

TreeMap для хранения элементов применяет красно-черное дерево. Красно-черное дерево это, частный случай двоичного дерева поиска.



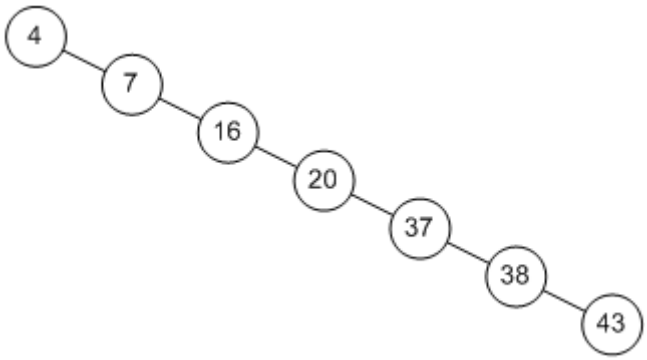
Двоичным деревом поиска (ДДП) называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя. Вершины, не имеющие потомков, называются листами. Подразумевается, что каждой вершине соответствует элемент или несколько элементов, имеющие некие ключевые значения, в дальнейшем именуемые просто ключами.

ДДП позволяет выполнять следующие основные операции:

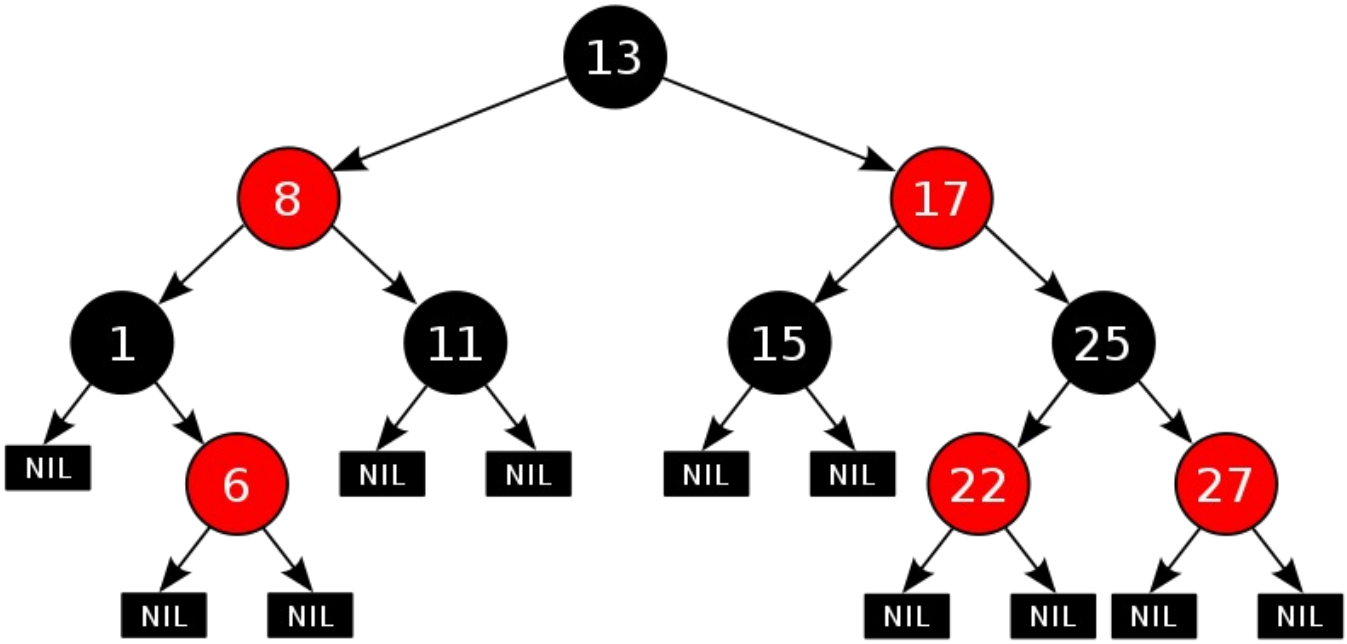
- Поиск вершины по ключу.
- Определение вершин с минимальным и максимальным значением ключа.
- Переход к предыдущей или последующей вершине, в порядке, определяемом ключами.
- Вставка вершины.
- Удаление вершины.

При реализации списком все функции требуют O(n) действий, где n — размер структуры. Операции с деревом же работают за O(h), где h — максимальная глубина дерева (глубина — расстояние от корня до вершины). В оптимальном случае, когда глубина всех листьев одинакова, в дереве будет n=2^h вершин. Значит, сложность операций в деревьях, близких к оптимуму будет O(log(n)).

В худшем случае дерево может следующий вид. Сложность операций в таком случае будет как у списка.



Для того, чтобы ситуации разложения в список не было, применяют специальные сбалансированные деревья. Среди них выделяют – B-деревья, AVL-деревья, Красно-чёрные деревья и т.д.



Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или черный. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно-чёрным деревьям применяются следующие требования:

1. Узел либо красный, либо чёрный.

2. Корень — чёрный. (В других определениях это правило иногда опускается. Это правило слабо влияет на анализ, так как корень всегда может быть изменен с красного на чёрный, но не обязательно наоборот).

3. Все листья(NIL) — черные.

4. Оба потомка каждого красного узла — черные.

5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов.

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html
3. https://habr.com/ru/post/128017/
4. https://habr.com/ru/post/421179/
5. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
6. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.

Вопросы для самоконтроля

1. В чем отличие Collection от массивов?

2. Я добавляю элемент в середину большого списка. На какой коллекции это будет быстрее- ArrayList или LinkedList?
3. Как производится балансировка в красно-черном дереве?
4. Какова алгоритмическая сложность поиска в хеш-коллекции?
5. Как возможно потерять элемент в hashMap?